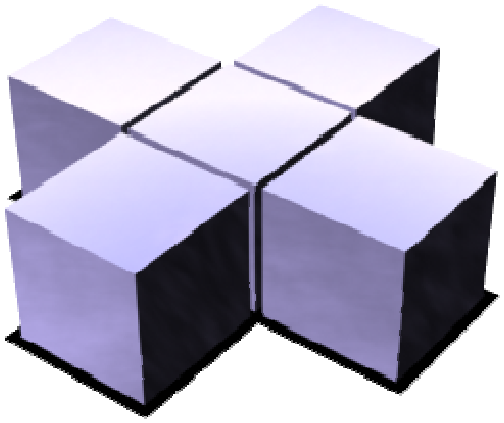
cplusplus.com

C++ Language Tutorial



Written by: **Juan Soulié**

Last revision: **June, 2007**

Available online at: http://www.cplusplus.com/doc/tutorial/

The online version is constantly revised and may contain corrections and changes

This document and its content is copyright of cplusplus.com © cplusplus.com, 2008. All rights reserved.

Any redistribution or reproduction of part or all of the content in any form is prohibited other than to print a personal copy of the entire document or download it to a local hard disk, without modifying its content in any way (including, but not limited to, this copyright notice).

You may not, except with express written permission from cplusplus.com, distribute the content of this document. Nor may you transmit it or store it in any other website or other form of electronic retrieval system.

© cplusplus.com 2008. All rights reserved

Table of contents

Table of contents **3**

Introduction **5**

Instructions for use 5

Basics of C++ **7**

Structure of a program 7

Variables. Data Types. 11

Constants 17

Operators 21

Basic Input/Output 29

Control Structures **34**

Control Structures 34

Functions (I) 41

Functions (II) 47

Compound data types **54**

Arrays 54

Character Sequences 60

Pointers 63

Dynamic Memory 74

Data structures 77

Other Data Types 82

Object Oriented Programming **86**

Classes (I) 86

Classes (II) 95

Friendship and inheritance 100

Polymorphism 107

Advanced concepts **113**

Templates 113

Namespaces 120

Exceptions 123

Type Casting 127

© cplusplus.com 2008. All rights reserved

Preprocessor directives 133

C++ Standard Library 138

Input/Output with files 138

© cplusplus.com 2008. All rights reserved

Introduction

Instructions for use

To whom is this tutorial directed?

This tutorial is for those people who want to learn programming in C++ and do not necessarily have any previous knowledge of other programming languages. Of course any knowledge of other programming languages or any general computer skill can be useful to better understand this tutorial, although it is not essential.

It is also suitable for those who need a little update on the new features the language has acquired from the latest standards.

If you are familiar with the C language, you can take the first 3 parts of this tutorial as a review of concepts, since they mainly explain the C part of C++. There are slight differences in the C++ syntax for some C features, so I recommend you its reading anyway.

The 4th part describes object-oriented programming.

The 5th part mostly describes the new features introduced by ANSI-C++ standard.

Structure of this tutorial

The tutorial is divided in 6 parts and each part is divided on its turn into different sections covering a topic each one. You can access any section directly from the section index available on the left side bar, or begin the tutorial from any point and follow the links at the bottom of each section.

Many sections include examples that describe the use of the newly acquired knowledge in the chapter. It is recommended to read these examples and to be able to understand each of the code lines that constitute it before passing to the next chapter.

A good way to gain experience with a programming language is by modifying and adding new functionalities on your own to the example programs that you fully understand. Don't be scared to modify the examples provided with this tutorial, that's the way to learn!

Compatibility Notes

The ANSI-C++ standard acceptation as an international standard is relatively recent. It was first published in November 1997, and revised in 2003. Nevertheless, the C++ language exists from a long time before (1980s). Therefore there are many compilers which do not support all the new capabilities included in ANSI-C++, especially those released prior to the publication of the standard.

This tutorial is thought to be followed with modern compilers that support -at least on some degree- ANSI-C++ specifications. I encourage you to get one if yours is not adapted. There are many options, both commercial and free.

Compilers

The examples included in this tutorial are all **console programs**. That means they use text to communicate with the user and to show their results.

© cplusplus.com 2008. All rights reserved

All C++ compilers support the compilation of console programs. Check the user's manual of your compiler for more info on how to compile them.

© cplusplus.com 2008. All rights reserved

Basics of C++

Structure of a program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

```
Hello World!
```

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

`// my first program in C++`

This is a comment line. All lines beginning with two slash signs (`//`) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

`#include <iostream>`

Lines beginning with a hash sign (`#`) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include <iostream>` tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

`using namespace std;`

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

`int main ()`

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a `main` function.

The word `main` is followed in the code by a pair of parentheses (`()`). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces (`{}`). What is contained within these braces is what the function does when it is executed.

© cplusplus.com 2008. All rights reserved

`cout << "Hello World!";`

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

`cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the `Hello World` sequence of characters) into the standard output stream (which usually is the screen).

`cout` is declared in the `iostream` standard file within the `std` namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (`;`). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

`return 0;`

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code `0`). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by `//`). There were lines with directives for the compiler's preprocessor (those beginning by `#`). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into `cout`), which were all included within the block delimited by the braces (`{}`) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
  cout << " Hello World!";
  return 0;
}
```

We could have written:

```
int main () { cout << "Hello World!"; return 0; }
```

All in just one line and this would have had exactly the same meaning as the previous code.

In C++, the separation between statements is specified with an ending semicolon (`;`) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

Let us add an additional instruction to our first program:

© cplusplus.com 2008. All rights reserved

```
// my second program in C++

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
  return 0;
}
```

```
Hello World! I'm a C++ program
```

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since `main` could have been perfectly valid defined this way:

```
int main () { cout << " Hello World! "; cout << " I'm a C++ program "; return 0; }
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
  cout <<
    "Hello World!";
  cout
    << "I'm a C++ program";
  return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by `#`) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (`;`).

Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment
/* block comment */
```

The first of them, known as line comment, discards everything from where the pair of slash signs (`//`) is found up to the end of that same line. The second one, known as block comment, discards everything between the `/*` characters and the first appearance of the `*/` characters, with the possibility of including more than one line.
We are going to add comments to our second program:

© cplusplus.com 2008. All rights reserved

```
/* my second program in C++
   with more comments */

#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";     // prints Hello
World!
  cout << "I'm a C++ program"; // prints I'm a
C++ program
  return 0;
}
```

```
Hello World! I'm a C++ program
```

If you include comments within the source code of your programs without using the comment characters combinations `//`, `/*` or `*/`, the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

© cplusplus.com 2008. All rights reserved

Variables. Data Types.

The usefulness of the "Hello World" programs shown in the previous section is quite questionable. We had to write several lines of code, compile them, and then execute the resulting program just to obtain a simple sentence written on the screen as result. It certainly would have been much faster to type the output sentence by ourselves. However, programming is not limited only to printing simple texts on the screen. In order to go a little further on and to become able to write programs that perform useful tasks that really save us work we need to introduce the concept of variable.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were `a`, `b` and `result`, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

Identifiers

A valid identifier is a sequence of one or more letters, digits or underscore characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but in some cases these may be reserved for compiler specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language nor your compiler's specific ones, which are *reserved keywords*. The standard reserved keywords are:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete,
do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto,
if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register,
reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template,
this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t, while
```

Additionally, alternative representations for some operators cannot be used as identifiers since they are reserved words under some circumstances:

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq
```


© cplusplus.com 2008. All rights reserved

Your compiler may also include some additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the `RESULT` variable is not the same as the `result` variable or the `Result` variable. These are three different variable identifiers.

Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int` (`short`)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int` (`long`)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	true or false
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`wchar_t`	Wide character.	2 *or* 4 bytes	1 wide character

* The values of the columns **Size** and **Range** depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that `int` has the natural size suggested by the system architecture (one "*word*") and the four integer types `char`, `short`, `int` and `long` must each one be at least as large as the one preceding it, with `char` being always 1 byte in size. The same applies to the floating point types `float`, `double` and `long double`, where each one must provide at least as much precision as the preceding one.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

© cplusplus.com 2008. All rights reserved

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type `int` with the identifier `a`. The second one declares a variable of type `float` with the identifier `mynumber`. Once declared, the variables `a` and `mynumber` can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (`a`, `b` and `c`), all of them of type `int`, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The integer data types `char`, `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name. For example:

```
unsigned short int NumberOfSisters;
signed int MyAccountBalance;
```

By default, if we do not specify either `signed` or `unsigned` most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword `signed`)

An exception to this general rule is the `char` type, which exists by itself and is considered a different fundamental data type from `signed char` and `unsigned char`, thought to store characters. You should use either `signed` or `unsigned` if you intend to store numerical values in a `char`-sized variable.

`short` and `long` can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: `short` is equivalent to `short int` and `long` is equivalent to `long int`. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, `signed` and `unsigned` may also be used as standalone type specifiers, meaning the same as `signed int` and `unsigned int` respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

© cplusplus.com 2008. All rights reserved

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

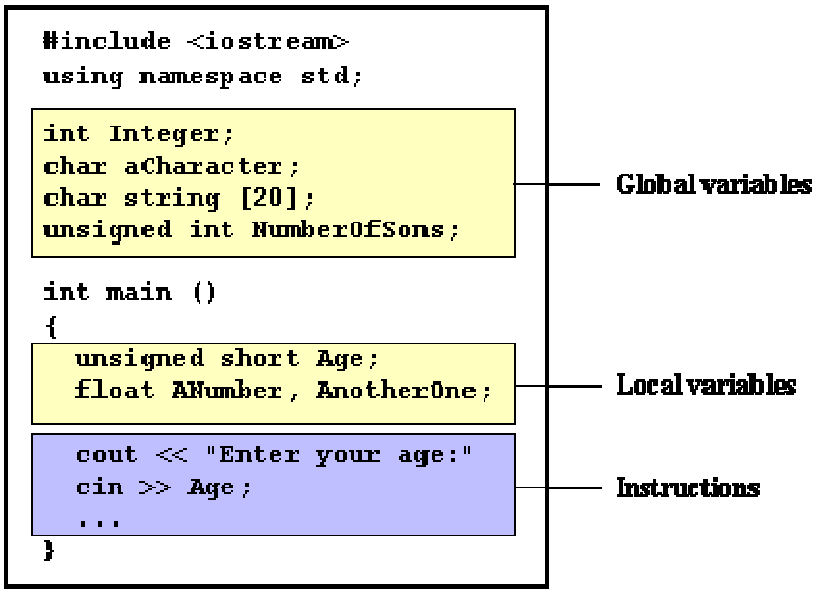
```
4
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that `a`, `b`, and `result` were of type `int`.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.



Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

© cplusplus.com 2008. All rights reserved

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function `main`) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to `main`, the local variables declared in `main` could not be accessed from the other function and vice versa.

Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (`()`):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // initial value = 5
  int b(2);              // initial value = 2
  int result;            // initial value
undetermined

  a = a + 3;
  result = a - b;
  cout << result;

  return 0;
}
```

```
6
```

Introduction to strings

Variables that can store non-numerical values that are longer than one single character are known as strings.

The C++ language library provides support for strings through the standard `string` class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage.

© cplusplus.com 2008. All rights reserved

A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: `<string>` and have access to the `std` namespace (which we already had in all our previous programs thanks to the `using namespace` statement).

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring = "This is a string";
  cout << mystring;
  return 0;
}
```

```
This is a string
```

As you may see in the previous example, strings can be initialized with any valid string literal just like numerical type variables can be initialized to any valid numerical literal. Both initialization formats are valid with strings:

```
string mystring = "This is a string";
string mystring ("This is a string");
```

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution:

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

```
This is the initial string content
This is a different string content
```

For more details on C++ strings, you can have a look at the string class reference.

© cplusplus.com 2008. All rights reserved

Constants

Constants are expressions with a fixed value.

Literals

Literals are used to express particular values within the source code of a program. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
a = 5;
```

the `5` in this piece of code was a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Integer Numerals

```
1776
707
-273
```

They are numerical constants that identify integer decimal values. Notice that to express a numerical constant we do not have to write quotes (") nor any special character. There is no doubt that it is a constant: whenever we write `1776` in a program, we will be referring to the value 1776.

In addition to decimal numbers (those that all of us are used to use every day) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a `0` (zero character). And in order to express a hexadecimal number we have to precede it with the characters `0x` (zero, x). For example, the following literal constants are all equivalent to each other:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type `int`. However, we can force them to either be unsigned by appending the `u` character to it, or long by appending `l`:

```
75          // int
75u         // unsigned int
75l         // long
75ul        // unsigned long
```

In both cases, the suffix can be specified using either upper or lowercase letters.

Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an `e` character (that expresses "by ten at the Xth height", where X is an integer value that follows the `e` character), or both a decimal point and an `e` character:

© cplusplus.com 2008. All rights reserved

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0         // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is `double`. If you explicitly want to express a `float` or `long double` numerical literal, you can use the `f` or `l` suffixes respectively:

```
3.14159L    // long double
6.02e23f    // float
```

Any of the letters that can be part of a floating-point numerical constant (`e`, `f`, `l`) can be written using either lower or uppercase letters without any difference in their meanings.

Character and string literals

There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

The first two expressions represent single character constants, and the following two represent string literals composed of several characters. Notice that to represent a single character we enclose it between single quotes (`'`) and to express a string (which generally consists of more than one character) we enclose it between double quotes (`"`).

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

```
x
'x'
```

`x` alone would refer to a variable whose identifier is `x`, whereas `'x'` (enclosed within single quotation marks) would refer to the character constant `'x'`.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (`\n`) or tab (`\t`). All of them are preceded by a backslash (`\`). Here you have a list of some of such escape codes:

© cplusplus.com 2008. All rights reserved

\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

```
"string expressed in \
two lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"this forms" "a single" "string" "of characters"
```

Finally, if we want the string literal to be explicitly made of wide characters (wchar_t), instead of narrow characters (char), we can precede the constant with the L prefix:

```
L"This is a wide character string"
```

Wide characters are used mainly to represent non-English or exotic character sets.

Boolean literals

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of type bool by using the Boolean literals true and false.

Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

© cplusplus.com 2008. All rights reserved

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: `PI` and `NEWLINE`. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;               // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

  return 0;
}
```

```
31.4159
```

In fact the only thing that the compiler preprocessor does when it encounters `#define` directives is to literally replace any occurrence of their identifier (in the previous example, these were `PI` and `NEWLINE`) by the code to which they have been defined (`3.14159` and `'\n'` respectively).

The `#define` directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (`;`) at its end. If you append a semicolon character (`;`) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

Declared constants (const)

With the `const` prefix you can declare constants with a specific type in the same way as you would do with a variable:

```
const int pathwidth = 100;
const char tabulator = '\t';
```

Here, `pathwidth` and `tabulator` are two typed constants. They are treated just like regular variables except that their values cannot be modified after their definition.

© cplusplus.com 2008. All rights reserved

Operators

Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

You do not have to memorize all the content of this page. Most details are only provided to serve as a later reference in case you need it.

Assignment (=)

The assignment operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable `a`. The part at the left of the assignment operator (`=`) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:

```
a = b;
```

This statement assigns to variable `a` (the lvalue) the value contained in variable `b` (the rvalue). The value that was stored until this moment in `a` is not considered at all in this operation, and in fact that value is lost.

Consider also that we are only assigning the value of `b` to `a` at the moment of the assignment operation. Therefore a later change of `b` will not affect the new value of `a`.

For example, let us have a look at the following code - I have included the evolution of the content stored in the variables as comments:

```
// assignment operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;         // a:?,  b:?
  a = 10;           // a:10, b:?
  b = 4;            // a:10, b:4
  a = b;            // a:4,  b:4
  b = 7;            // a:4,  b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

```
a:4 b:7
```

This code will give us as result that the value contained in `a` is `4` and the one contained in `b` is `7`. Notice how `a` was not affected by the final modification of `b`, even though we declared `a = b` earlier (that is because of the *right-to-left rule*).

© cplusplus.com 2008. All rights reserved

A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
a = 2 + b;
```

that means: first assign `5` to variable `b` and then assign to `a` the value `2` plus the result of the previous assignment of `b` (i.e. 5), leaving `a` with a final value of `7`.

The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns `5` to the all the three variables: `a`, `b` and `c`.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see is *modulo*; whose operator is the percentage sign (`%`). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable `a` will contain the value `2`, since `2` is the remainder from dividing `11` between `3`.

Compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

expression	is equivalent to
`value += increase;`	`value = value + increase;`
`a -= 5;`	`a = a - 5;`
`a /= b;`	`a = a / b;`
`price *= units + 1;`	`price = price * (units + 1);`

and the same for all other operators. For example:

© cplusplus.com 2008. All rights reserved

```
// compound assignment operators

#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;              // equivalent to a=a+2
  cout << a;
  return 0;
}
```

```
5
```

Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (`++`) and the decrease operator (`--`) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. Thus:

```
c++;
c+=1;
c=c+1;
```

are all equivalent in its functionality: the three of them increase by one the value of c.

In the early C compilers, the three previous expressions probably produced different executable code depending on which one was used. Nowadays, this type of code optimization is generally done automatically by the compiler, thus the three expressions should produce exactly the same executable code.

A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (`++a`) or after it (`a++`). Although in simple expressions like `a++` or `++a` both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (`a++`) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. Notice the difference:

Example 1	Example 2

`B=3;`
`A=++B;`
`// A contains 4, B contains 4`
	`B=3;`
	`A=B++;`
	`// A contains 3, B contains 4`

In Example 1, `B` is increased before its value is copied to `A`. While in Example 2, the value of `B` is copied to `A` and then `B` is increased.

Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

© cplusplus.com 2008. All rights reserved

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Here there are some examples:

```
(7 == 5)     // evaluates to false.
(5 > 4)      // evaluates to true.
(3 != 2)     // evaluates to true.
(6 >= 6)     // evaluates to true.
(5 < 5)      // evaluates to false.
```

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that `a=2`, `b=3` and `c=6`,

```
(a == 5)     // evaluates to false since a is not equal to 5.
(a*b >= c)   // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c)  // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.
```

Be careful! The operator `=` (one equal sign) is not the same as the operator `==` (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (`==`) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression `((b=2) == a)`, we first assigned the value `2` to `b` and then we compared it to `a`, that also stores the value `2`, so the result of the operation is true.

Logical operators (!, &&, ||)

The Operator `!` is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)     // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)     // evaluates to true because (6 <= 4) would be false.
!true         // evaluates to false
!false        // evaluates to true.
```

The logical operators `&&` and `||` are used when evaluating two expressions to obtain a single relational result. The operator `&&` corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator `&&` evaluating the expression `a && b`:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator `||` corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of `a || b`:

© cplusplus.com 2008. All rights reserved

|| OPERATOR

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

Conditional operator (?)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:

```
condition ? result1 : result2
```

If `condition` is true the expression will return `result1`, if it is not it will return `result2`.

```
7==5 ? 4 : 3      // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3    // returns 4, since 7 is equal to 5+2.
5>3 ? a : b       // returns the value of a, since 5 is greater than 3.
a>b ? a : b       // returns whichever is greater, a or b.
```

```
// conditional operator

#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c;

  return 0;
}
```

```
7
```

In this example `a` was `2` and `b` was `7`, so the expression being evaluated (`a>b`) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was `b`, with a value of `7`.

Comma operator (,)

The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

For example, the following code:

```
a = (b=3, b+2);
```


© cplusplus.com 2008. All rights reserved

Would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

Explicit type casting operator

Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

sizeof()

This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type.
The value returned by sizeof is a constant, so it is always determined before program execution.

Other operators

Later in these tutorials, we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```


© cplusplus.com 2008. All rights reserved

we may doubt if it really means:

```
a = 5 + (7 % 2)     // with a result of 6, or
a = (5 + 7) % 2     // with a result of 0
```

The correct answer is the first of the two expressions, with a result of `6`. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Grouping
1	`::`	scope	Left-to-right
2	`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	postfix	Left-to-right
3	`++ -- ~ ! sizeof new delete`	unary (prefix)	Right-to-left
	`* &`	indirection and reference (pointers)	
	`+ -`	unary sign operator	
4	`(type)`	type casting	Right-to-left
5	`.* ->*`	pointer-to-member	Left-to-right
6	`* / %`	multiplicative	Left-to-right
7	`+ -`	additive	Left-to-right
8	`<< >>`	shift	Left-to-right
9	`< > <= >=`	relational	Left-to-right
10	`== !=`	equality	Left-to-right
11	`&`	bitwise AND	Left-to-right
12	`^`	bitwise XOR	Left-to-right
13	`|`	bitwise OR	Left-to-right
14	`&&`	logical AND	Left-to-right
15	`||`	logical OR	Left-to-right
16	`?:`	conditional	Right-to-left
17	`= *= /= %= += -= >>= <<= &= ^= |=`	assignment	Right-to-left
18	`,`	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression.

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs `(` and `)`, as in this example:

```
a = 5 + 7 % 2;
```


© cplusplus.com 2008. All rights reserved

might be written either as:

```
a = 5 + (7 % 2);
```

or

```
a = (5 + 7) % 2;
```

depending on the operation that we want to perform.

So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.

© cplusplus.com 2008. All rights reserved

Basic Input/Output

Until now, the example programs of previous sections provided very little interaction with the user, if any at all. Using the standard input and output library, we will be able to interact with the user by printing messages on the screen and getting the user's input from the keyboard.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is `cout`.

`cout` is used in conjunction with the *insertion operator*, which is written as `<<` (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints the content of x on screen
```

The `<<` operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string `Output sentence`, the numerical constant `120` and variable `x` into the standard output stream `cout`. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";  // prints Hello
cout << Hello;    // prints the content of Hello variable
```

The insertion operator (`<<`) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message `Hello, I am a C++ statement` on the screen. The utility of repeating the insertion operator (`<<`) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the `age` variable to contain the value `24` and the `zipcode` variable to contain `90064` the output of the previous statement would be:

```
Hello, I am 24 years old and my zipcode is 90064
```

It is important to notice that `cout` does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

© cplusplus.com 2008. All rights reserved

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

```
This is a sentence.This is another sentence.
```

even though we had written them in two different insertions into `cout`. In order to perform a line break on the output we must explicitly insert a new-line character into `cout`. In C++ a new-line character can be specified as `\n` (backslash, n):

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Additionally, to add a new-line, you may also use the `endl` manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

```
First sentence.
Second sentence.
```

The `endl` manipulator produces a newline character, exactly as the insertion of `'\n'` does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, `cout` will be an unbuffered stream in most cases, so you can generally use both the `\n` escape character and the `endl` manipulator in order to specify a new line without any difference in its behavior.

Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (`>>`) on the `cin` stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type `int` called `age`, and the second one waits for an input from `cin` (the keyboard) in order to store it in this integer variable.

`cin` can only process the input from the keyboard once the `RETURN` key has been pressed. Therefore, even if you request a single character, the extraction from `cin` will not process the input until the user presses `RETURN` after the character has been introduced.

You must always consider the type of the variable that you are using as a container with `cin` extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

© cplusplus.com 2008. All rights reserved

```
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its double is
1404.
```

The user of a program may be one of the factors that generate errors even in the simplest programs that use `cin` (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by `cin` extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. A little ahead, when we see the `stringstream` class we will see a possible solution for the errors that can be caused by this type of user input.

You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

In both cases the user must give two data, one for variable `a` and another one for variable `b` that may be separated by any valid blank separator: a space, a tab character or a newline.

cin and strings

We can use `cin` to get strings with the extraction operator (`>>`) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, `cin` extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction. This behavior may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful.

In order to get entire lines, we can use the function `getline`, which is the more recommendable way to get user input with `cin`:

© cplusplus.com 2008. All rights reserved

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

```
What's your name? Juan SouliÃ¯Â¿Â½
Hello Juan SouliÃ¯Â¿Â½.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

Notice how in both calls to `getline` we used the same string identifier (`mystr`). What the program does in the second call is simply to replace the previous content by the new one that is introduced.

stringstream

The standard header file `<sstream>` defines a class called `stringstream` that allows a string-based object to be treated as a stream. This way we can perform extraction or insertion operations from/to strings, which is especially useful to convert strings to numerical values and vice versa. For example, if we want to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

This declares a `string` object with a value of `"1204"`, and an `int` object. Then we use `stringstream`'s constructor to construct an object of this type from the string object. Because we can use `stringstream` objects as if they were streams, we can extract an integer from it as we would have done on `cin` by applying the extractor operator (`>>`) on it followed by a variable of type `int`.

After this piece of code, the variable `myint` will contain the numerical value `1204`.

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
  string mystr;
  float price=0;
  int quantity=0;

  cout << "Enter price: ";
  getline (cin,mystr);
  stringstream(mystr) >> price;
  cout << "Enter quantity: ";
  getline (cin,mystr);
  stringstream(mystr) >> quantity;
  cout << "Total price: " << price*quantity <<
endl;
  return 0;
}
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

In this example, we acquire numeric values from the standard input indirectly. Instead of extracting numeric values directly from the standard input, we get lines from the standard input (`cin`) into a string object (`mystr`), and then we extract the integer values from this string into a variable of type int (`quantity`).

© cplusplus.com 2008. All rights reserved

Using this method, instead of direct extractions of integer values, we have more control over what happens with the input of numeric values from the user, since we are separating the process of obtaining input from the user (we now simply ask for lines) with the interpretation of that input. Therefore, this method is usually preferred to get numerical values from the user in all programs that are intensive in user input.

© cplusplus.com 2008. All rights reserved

Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: `{ }`:

```
{ statement1; statement2; statement3; }
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces (`{}`). But in the case that we want the statement to be a compound statement it must be enclosed between braces (`{}`), forming a block.

Conditional structure: if and else

The `if` keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition) statement
```

Where `condition` is the expression that is being evaluated. If this condition is true, `statement` is executed. If it is false, `statement` is ignored (not executed) and the program continues right after this conditional structure.
For example, the following code fragment prints `x is 100` only if the value stored in the `x` variable is indeed `100`:

```
if (x == 100)
  cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces `{ }`:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword `else`. Its form used in conjunction with `if` is:

```
if (condition) statement1 else statement2
```

For example:

© cplusplus.com 2008. All rights reserved

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

prints on the screen `x is 100` if indeed `x` has a value of `100`, but if it has not -and only if not- it prints out `x is not 100`.

The `if + else` structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in `x` is positive, negative or none of them (i.e. zero):

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces `{ }`.

Iteration structures (loops)

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

```
while (expression) statement
```

and its functionality is simply to repeat statement while the condition set in expression is true.
For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while

#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "FIRE!\n";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the `while` loop begins, if the value entered by the user fulfills the condition `n>0` (that `n` is greater than zero) the block that follows the condition will be executed and repeated while the condition (`n>0`) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

© cplusplus.com 2008. All rights reserved

1. User assigns a value to `n`
2. The while condition is checked (`n>0`). At this point there are two posibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   `cout << n << ", ";`
   `--n;`
   (prints the value of `n` on the screen and decreases `n` by `1`)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included `--n;` that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) to become false after a certain number of loop iterations: to be more specific, when `n` becomes `0`, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

```
do statement while (condition);
```

Its functionality is exactly the same as the while loop, except that `condition` in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of `statement` even if `condition` is never fulfilled. For example, the following example program echoes any number you enter until you enter `0`.

```
// number echoer

#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value `0` in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

```
for (initialization; condition; increase) statement;
```


© cplusplus.com 2008. All rights reserved

and its main function is to repeat `statement` while `condition` remains true, like the while loop. But in addition, the `for` loop provides specific locations to contain an `initialization` statement and an `increase` statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. `initialization` is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. `condition` is checked. If it is true the loop continues, otherwise the loop ends and `statement` is skipped (not executed).
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in braces `{ }`.
4. finally, whatever is specified in the `increase` field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

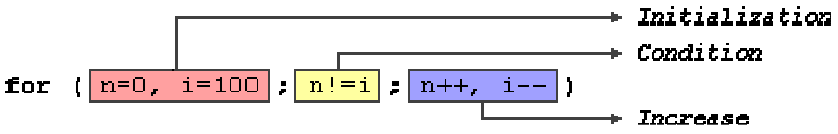
```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The `initialization` and `increase` fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: `for (;n<10;)` if we wanted to specify no initialization and no increase; or `for (;n<10;n++)` if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a `for` loop, like in `initialization`, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // whatever here...
}
```

This loop will execute for 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of `0`, and `i` with `100`, the condition is `n!=i` (that `n` is not equal to `i`). Because `n` is increased by one and `i` decreased by one, the loop's condition will become false after the 50th loop, when both `n` and `i` will be equal to `50`.

© cplusplus.com 2008. All rights reserved

Jump statements.

The break statement

Using `break` we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example

#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!
```

The goto statement

`goto` allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.
The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (`:`).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using `goto`:

© cplusplus.com 2008. All rights reserved

```
// goto loop example

#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!\n";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The exit function

`exit` is a function defined in the `cstdlib` library.

The purpose of `exit` is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of `0` means that the program finished normally and any other value means that some error or unexpected results happened.

The selective structure: switch.

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several `if` and `else if` instructions. Its form is the following:

```
switch (expression)
{
  case constant1:
     group of statements 1;
     break;
  case constant2:
     group of statements 2;
     break;
  .
  .
  .
  default:
     default group of statements
}
```

It works in the following way: switch evaluates `expression` and checks if it is equivalent to `constant1`, if it is, it executes `group of statements 1` until it finds the `break` statement. When it finds this `break` statement the program jumps to the end of the `switch` selective structure.

If expression was not equal to `constant1` it will be checked against `constant2`. If it is equal to this, it will execute `group of statements 2` until a break keyword is found, and then will jump to the end of the `switch` selective structure.

Finally, if the value of `expression` did not match any of the previously specified constants (you can include as many `case` labels as values you want to check), the program will execute the statements included after the `default:` label, if it exists (since it is optional).

© cplusplus.com 2008. All rights reserved

Both of the following code fragments have the same behavior:

switch example	if-else equivalent

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`if (x == 1) {`
	`  cout << "x is 1";`
	`  }`
	`else if (x == 2) {`
	`  cout << "x is 2";`
	`  }`
	`else {`
	`  cout << "value of x unknown";`
	`  }`

The `switch` statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put `break` statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the `switch` selective block or a `break` statement is reached.

For example, if we did not include a `break` statement after the first group for case one, the program will not automatically jump to the end of the `switch` selective block and it would continue executing the rest of statements until it reaches either a `break` instruction or the end of the `switch` selective block. This makes unnecessary to include braces `{ }` surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example `case n:` where `n` is a variable) or ranges (`case (1..3):`) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of `if` and `else if` statements.

© cplusplus.com 2008. All rights reserved

Functions (I)

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

```
type name ( parameter1, parameter2, ...) { statements }
```

where:

- `type` is the data type specifier of the data returned by the function.
- `name` is the identifier by which it will be possible to call the function.
- `parameters` (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- `statements` is the function's body. It is a block of statements surrounded by braces `{ }`.

Here you have the first function example:

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

```
The result is 8
```

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the `main` function. So we will begin there.

We can see how the `main` function begins by declaring the variable `z` of type `int`. Right after that, we see a call to a function called `addition`. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)
               ^       ^
z = addition (  5  ,   3  );
```

The parameters and arguments have a clear correspondence. Within the `main` function we called to `addition` passing two values: `5` and `3`, that correspond to the `int a` and `int b` parameters declared for function addition.

© cplusplus.com 2008. All rights reserved

At the point at which the function is called from within `main`, the control is lost by `main` and passed to function `addition`. The value of both arguments passed in the call (`5` and `3`) are copied to the local variables `int a` and `int b` within the function.

Function `addition` declares another local variable (`int r`), and by means of the expression `r=a+b`, it assigns to `r` the result of `a` plus `b`. Because the actual parameters passed for `a` and `b` are `5` and `3` respectively, the result is `8`.

The following line of code:

```
return (r);
```

finalizes function `addition`, and returns the control back to the function that called it in the first place (in this case, `main`). At this moment the program follows it regular course from the same point at which it was interrupted by the call to `addition`. But additionally, because the `return` statement in function `addition` specified a value: the content of variable `r` (`return (r);`), which at that moment had a value of `8`. This value becomes the value of evaluating the function call.

```
int addition (int a, int b)
 |8
z = addition (  5  ,   3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable `z` will be set to the value returned by `addition (5, 3)`, that is `8`. To explain it another way, you can imagine that the call to a function (`addition (5,3)`) is literally replaced by the value it returns (`8`).

The following line of code in main is:

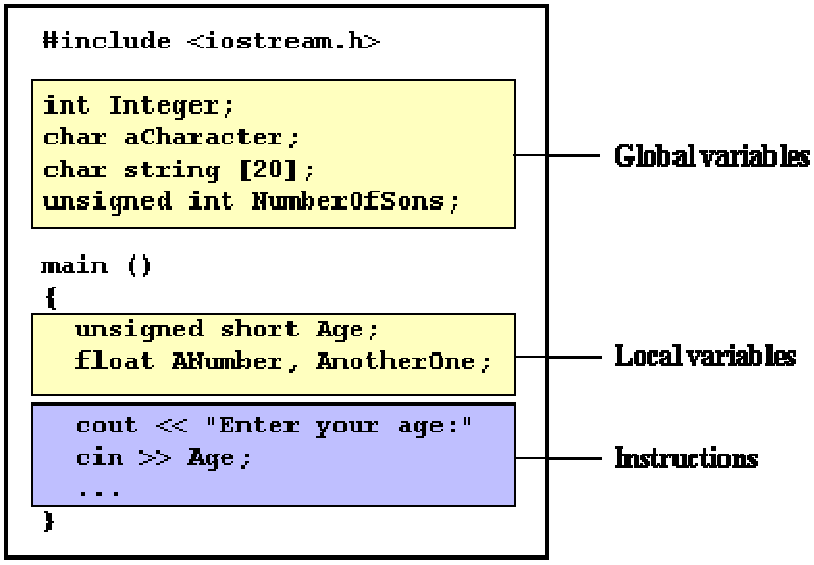
```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

© cplusplus.com 2008. All rights reserved

Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables `a`, `b` or `r` directly in function `main` since they were variables local to function `addition`. Also, it would have been impossible to use the variable `z` directly within function `addition`, since this was a variable local to the function `main`.



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

© cplusplus.com 2008. All rights reserved

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

In this case we have created a function called `subtraction`. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function `main` we will see that we have made several calls to function `subtraction`. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., `5`), we would have:

```
z = 5;
cout << "The first result is " << z;
```

As well as

```
cout << "The second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to `subtraction` directly as an insertion parameter for `cout`. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since `5` is the value returned by `subtraction (7,2)`.

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```


© cplusplus.com 2008. All rights reserved

The only new thing that we introduced is that the parameters of `subtraction` are variables instead of constants. That is perfectly valid. In this case the values passed to function `subtraction` are the values of `x` and `y`, that are `5` and `3` respectively, giving `2` as result.

The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction (x,y);
```

we could have written:

```
z = subtraction (x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (`;`) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:

```
z = 4 + 2;
z = 2 + 4;
```

Functions with no type. The use of void.

If you remember the syntax of a function declaration:

```
type name ( argument1, argument2 ...) statement
```

you will see that the declaration begins with a `type`, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the `void` type specifier for the function. This is a special specifier that indicates absence of type.

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
  return 0;
}
```

```
I'm a function!
```

`void` can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function `printmessage` could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

Although it is optional to specify `void` in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

© cplusplus.com 2008. All rights reserved

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to `printmessage` is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

```
printmessage;
```


© cplusplus.com 2008. All rights reserved

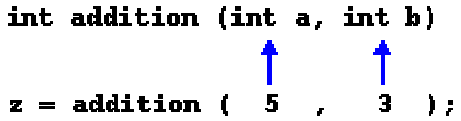
Functions (II)

Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function `addition` using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of `x` and `y`, i.e. `5` and `3` respectively, but not the variables `x` and `y` themselves.



This way, when the function addition is called, the value of its local variables `a` and `b` become `5` and `3` respectively, but any modification to either `a` or `b` within the function addition will not have any effect in the values of `x` and `y` outside it, because variables `x` and `y` were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

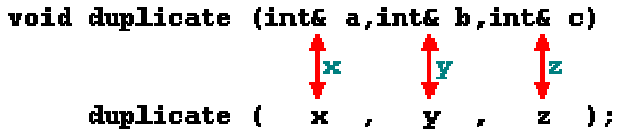
```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

```
x=2, y=6, z=14
```

The first thing that should call your attention is that in the declaration of `duplicate` the type of each parameter was followed by an ampersand sign (`&`). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

© cplusplus.com 2008. All rights reserved



To explain it in another way, we associate `a`, `b` and `c` with the arguments passed on the function call (`x`, `y` and `z`) and any change that we do on `a` within the function will affect the value of `x` outside it. Any change that we do on `b` will affect `y`, and the same with `c` and `z`.

That is why our program's output, that shows the values stored in `x`, `y` and `z` after the call to `duplicate`, shows the values of all the three variables of `main` doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (`&`), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of `x`, `y` and `z` without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// more than one returning value
#include <iostream>
using namespace std;

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

```
Previous=99, Next=101
```

Default values in parameters.

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

© cplusplus.com 2008. All rights reserved

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

```
6
5
```

As we can see in the body of the program there are two calls to function `divide`. In the first one:

```
divide (12)
```

we have only specified one argument, but the function `divide` allows up to two. So the function `divide` has assumed that the second parameter is `2` since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is `6` (`12/2`).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for `b` (`int b=2`) is ignored and `b` takes the value passed as argument, that is `4`, making the result returned equal to `5` (`20/4`).

© cplusplus.com 2008. All rights reserved

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
// overloaded function
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

float operate (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << operate (x,y);
  cout << "\n";
  cout << operate (n,m);
  cout << "\n";
  return 0;
}
```

```
10
2.5
```

In this case we have defined two functions with the same name, `operate`, but one of them accepts two parameters of type `int` and the other one accepts them of type `float`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two `int` parameters in its prototype and if it is called with two floats it will call to the one which has two `float` parameters in its prototype.

In the first call to `operate` the two arguments passed are of type `int`, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type `float`, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to `operate` depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

inline functions.

The `inline` specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the `inline` keyword when calling the function, only in its declaration.

© cplusplus.com 2008. All rights reserved

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

more concretely, 5! (factorial of 5) would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return (1);
}

int main ()
{
  long number;
  cout << "Please type a number: ";
  cin >> number;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

```
Please type a number: 9
9! = 362880
```

Notice how in function `factorial` we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to `0` it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (`long`) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function `main` which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function `main` before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

© cplusplus.com 2008. All rights reserved

```
type name ( argument_type1, argument_type2, ...);
```

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces `{ }`) and instead of that we end the prototype declaration with a mandatory semicolon (`;`).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called `protofunction` with two `int` parameters with any of the following declarations:

```
int protofunction (int first, int second);
int protofunction (int, int);
```

Anyway, including a name for each variable makes the prototype more legible.

```
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int a);
void even (int a);

int main ()
{
  int i;
  do {
    cout << "Type a number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}

void even (int a)
{
  if ((a%2)==0) cout << "Number is even.\n";
  else odd (a);
}
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions `odd` and `even`:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in `main`, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in `odd` there is a call to `even` and in `even` there is a call to `odd`. If none of the two functions had been

© cplusplus.com 2008. All rights reserved

previously declared, a compilation error would happen, since either `odd` would not not be visible from `even` (because it has still not been declared), or `even` would not be visible from `odd` (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

© cplusplus.com 2008. All rights reserved

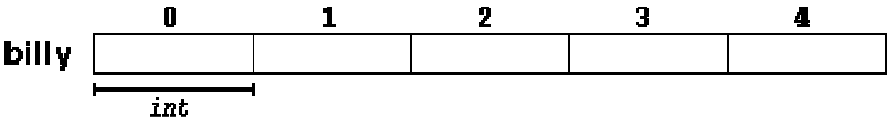
Compound data types

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, we can store 5 values of type `int` in an array without having to declare 5 different variables, each one with a different identifier. Instead of that, using an array we can store 5 different values of the same type, `int` for example, with a unique identifier.

For example, an array to contain 5 integer values of type `int` called `billy` could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type `int`. These elements are numbered from `0` to `4` since in arrays the first index is always `0`, independently of its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where `type` is a valid type (like `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain.

Therefore, in order to declare an array called `billy` as the one shown in the above diagram it is as simple as:

```
int billy [5];
```

NOTE: The `elements` field within brackets `[]` which represents the number of elements the array is going to hold, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained later in these tutorials.

Initializing arrays.

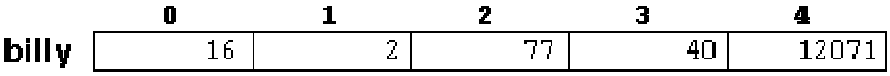
When declaring a regular array of local scope (within a function, for example), if we do not specify otherwise, its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, on the other hand, are automatically initialized with their default values, which for all fundamental types this means they are filled with zeros.

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each one of its elements by enclosing the values in braces `{ }`. For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

This declaration would have created an array like this:

© cplusplus.com 2008. All rights reserved



The amount of values between braces { } must not be larger than the number of elements that we declare for the array between square brackets [ ]. For example, in the example of array billy we have declared that it has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }:

```
int billy [] = { 16, 2, 77, 40, 12071 };
```

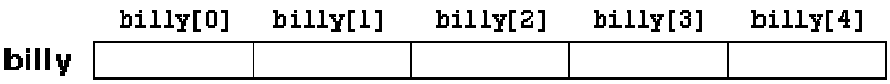
After this declaration, array billy would be 5 ints long, since we have provided 5 initialization values.

Accessing the values of an array.

In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

```
name[index]
```

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:



For example, to store the value 75 in the third element of billy, we could write the following statement:

```
billy[2] = 75;
```

and, for example, to pass the value of the third element of billy to a variable called a, we could write:

```
a = billy[2];
```

Therefore, the expression billy[2] is for all purposes like a variable of type int.

Notice that the third element of billy is specified billy[2], since the first one is billy[0], the second one is billy[1], and therefore, the third one is billy[2]. By this same reason, its last element is billy[4]. Therefore, if we write billy[5], we would be accessing the sixth element of billy and therefore exceeding the size of the array.

In C++ it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors. The reason why this is allowed will be seen further ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements. Do not confuse these two possible uses of brackets [ ] with arrays.

© cplusplus.com 2008. All rights reserved

```
int billy[5];          // declaration of a new array
billy[2] = 75;         // access to an element of the array.
```

If you read carefully, you will see that a type specifier always precedes a variable or array declaration, while it never precedes an access.

Some other valid operations with arrays:

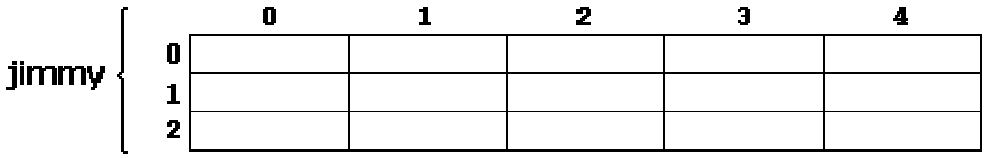
```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// arrays example
#include <iostream>
using namespace std;

int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    result += billy[n];
  }
  cout << result;
  return 0;
}
```

```
12206
```

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of them of a same uniform data type.



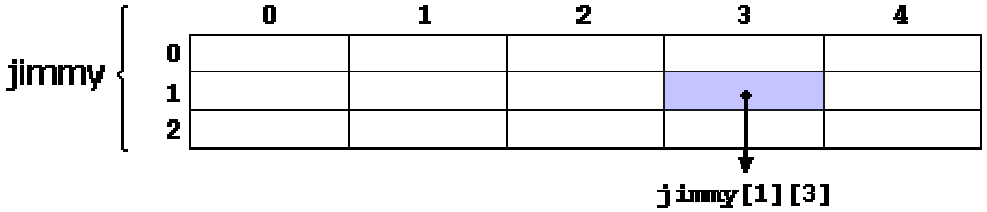
`jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The way to declare this array in C++ would be:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```


© cplusplus.com 2008. All rights reserved



(remember that array indices always begin by zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. But be careful! The amount of memory needed for an array rapidly increases with each dimension. For example:

```
char century [100][365][24][60][60];
```

declares an array with a `char` element for each second in a century, that is more than 3 billion chars. So this declaration would consume more than 3 gigabytes of memory!

Multidimensional arrays are just an abstraction for programmers, since we can obtain the same results with a simple array just by putting a factor between its indices:

```
int jimmy [3][5];   // is equivalent to
int jimmy [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays the compiler remembers the depth of each imaginary dimension for us. Take as example these two pieces of code, with both exactly the same result. One uses a bidimensional array and the other one uses a simple array:

multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

pseudo-multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
  return 0;
}
```

None of the two source codes above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

© cplusplus.com 2008. All rights reserved

We have used "defined constants" (`#define`) to simplify possible future modifications of the program. For example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done simply by changing the line:

```
#define HEIGHT 3
```

to:

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets `[]`. For example, the following function:

```
void procedure (int arg[])
```

accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;
}
```

```
5 10 15
2 4 6 8 10
```

As you can see, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that

© cplusplus.com 2008. All rights reserved

we pass to it as its first parameter. This allows the `for` loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for novice programmers. I recommend the reading of the chapter about Pointers for a better understanding on how arrays operate.

© cplusplus.com 2008. All rights reserved

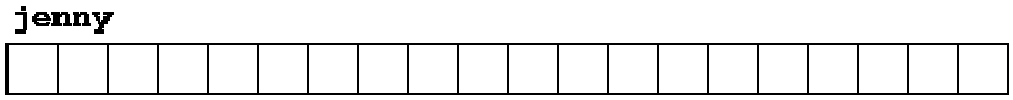
Character Sequences

As you may already know, the C++ Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters. However, because strings are in fact sequences of characters, we can represent them also as plain arrays of `char` elements.

For example, the following array:

```
char jenny [20];
```

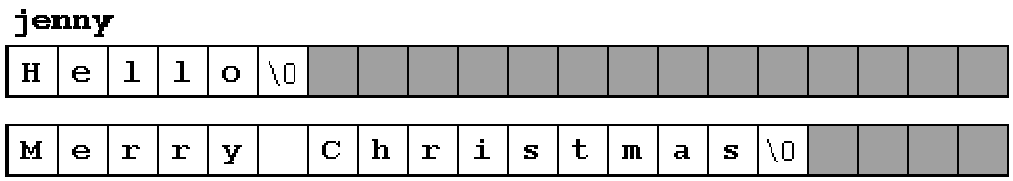
is an array that can store up to 20 elements of type `char`. It can be represented as:



Therefore, in this array, in theory, we can store sequences of characters up to 20 characters long. But we can also store shorter sequences. For example, `jenny` could store at some point in a program either the sequence `"Hello"` or the sequence `"Merry christmas"`, since both are shorter than 20 characters.

Therefore, since the array of characters can store shorter sequences than its total length, a special character is used to signal the end of the valid sequence: the *null character*, whose literal constant can be written as `'\0'` (backslash, zero).

Our array of 20 elements of type `char`, called `jenny`, can be represented storing the characters sequences `"Hello"` and `"Merry Christmas"` as:



Notice how after the valid content a null character (`'\0'`) has been included in order to indicate the end of the sequence. The panels in gray color represent `char` elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays they follow all their same rules. For example, if we want to initialize an array of characters with some predetermined sequence of characters we can do it just like any other array:

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared an array of 6 elements of type `char` initialized with the characters that form the word `"Hello"` plus a null character `'\0'` at the end.
But arrays of `char` elements have an additional method to initialize their values: using string literals.

In the expressions we have used in some examples in previous chapters, constants that represent entire strings of characters have already showed up several times. These are specified enclosing the text to become a string literal between double quotes ("). For example:

```
"the result is: "
```


© cplusplus.com 2008. All rights reserved

is a constant string literal that we have probably used already.

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we can initialize the array of char elements called myword with a null-terminated sequence of characters by either one of these two methods:

```
char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char myword [] = "Hello";
```

In both cases the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello" plus a final null character ('\0') which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that we are talking about initializing an array of characters in the moment it is being declared, and not about assigning values to them once they have already been declared. In fact because this type of null-terminated arrays of characters are regular arrays we have the same restrictions that we have with any other array, so we are not able to copy blocks of data with an assignment operation.

Assuming mystext is a char[] variable, expressions within a source code like:

```
mystext = "Hello";
mystext[] = "Hello";
```

would not be valid, like neither would be:

```
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The reason for this may become more comprehensible once you know a bit more about pointers, since then it will be clarified that an array is in fact a constant pointer pointing to a block of memory.

Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. In fact, regular string literals have this type (char[]) and can also be used in most cases.

For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

© cplusplus.com 2008. All rights reserved

```
// null-terminated sequences of characters
#include <iostream>
using namespace std;

int main ()
{
  char question[] = "Please, enter your first
name: ";
  char greeting[] = "Hello, ";
  char yourname [80];
  cout << question;
  cin >> yourname;
  cout << greeting << yourname << "!";
  return 0;
}
```

```
Please, enter your first name: John
Hello, John!
```

As you can see, we have declared three arrays of `char` elements. The first two were initialized with string literal constants, while the third one was left uninitialized. In any case, we have to speficify the size of the array: in the first two (`question` and `greeting`) the size was implicitly defined by the length of the literal constant they were initialized to. While for `yourname` we have explicitly specified that it has a size of 80 chars.

Finally, sequences of characters stored in `char` arrays can easily be converted into `string` objects just by using the assignment operator:

```
string mystring;
char myntcs[]="some text";
mystring = myntcs;
```


© cplusplus.com 2008. All rights reserved

Pointers

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

Reference operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (`&`), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

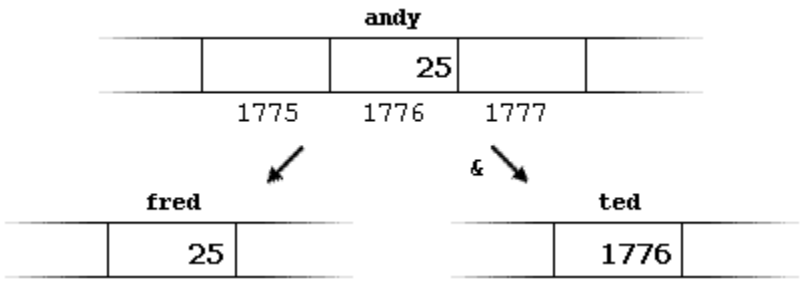
This would assign to `ted` the address of variable `andy`, since when preceding the name of the variable `andy` with the reference operator (`&`) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that `andy` is placed during runtime in the memory address `1776`. This number (`1776`) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
andy = 25;
fred = andy;
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



First, we have assigned the value 25 to `andy` (a variable whose address in memory we have assumed to be 1776).

© cplusplus.com 2008. All rights reserved

The second statement copied to `fred` the content of variable `andy` (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to `ted` not the value contained in `andy` but a reference to it (i.e., its address, which we have assumed to be `1776`). The reason is that in this third assignment operation we have preceded the identifier `andy` with the reference operator (`&`), so we were no longer referring to the value of andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like `ted` in the previous example) is what we call a *pointer*. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

Dereference operator (*)

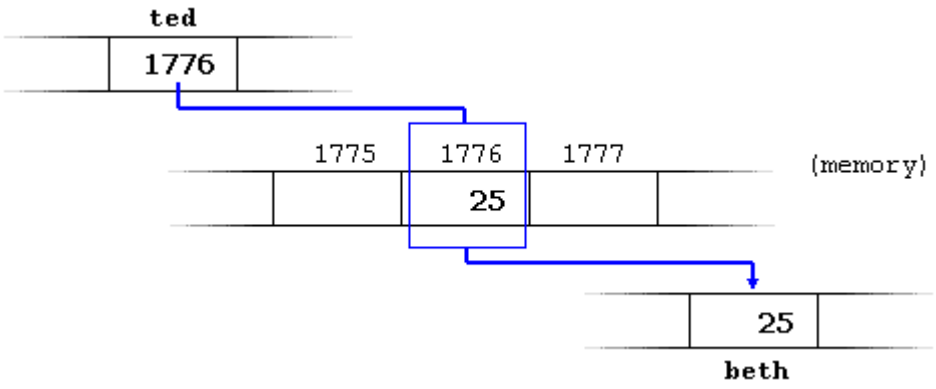
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "`beth` equal to value pointed by `ted`") `beth` would take the value `25`, since `ted` is `1776`, and the value pointed by 1776 is 25.



You must clearly differentiate that the expression `ted` refers to the value `1776`, while `*ted` (with an asterisk `*` preceding the identifier) refers to the value stored at address `1776`, which in this case is `25`. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

```
beth = ted;   // beth equal to ted ( 1776 )
beth = *ted;  // beth equal to value pointed by ted ( 25 )
```

Notice the difference between the reference and dereference operators:

- & is the reference operator and can be read as "address of"
- * is the dereference operator and can be read as "value pointed by"

© cplusplus.com 2008. All rights reserved

Thus, they have complementary (or opposite) meanings. A variable referenced with `&` can be dereferenced with `*`.

Earlier we performed the following two assignment operations:

```
andy = 25;
ted = &andy;
```

Right after these two statements, all of the following expressions would give true as result:

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

The first expression is quite clear considering that the assignment operation performed on `andy` was `andy=25`. The second one uses the reference operator (`&`), which returns the address of variable `andy`, which we assumed it to have a value of `1776`. The third one is somewhat obvious since the second expression was true and the assignment operation performed on `ted` was `ted=&andy`. The fourth expression uses the dereference operator (`*`) that, as we have just seen, can be read as "value pointed by", and the value pointed by `ted` is indeed `25`.

So, after all that, you may also infer that for as long as the address pointed by `ted` remains unchanged the following expression will also be true:

```
*ted == andy
```

Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a `char` as to point to an `int` or a `float`.

The declaration of pointers follows this format:

```
type * name;
```

where `type` is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:

```
int * number;
char * character;
float * greatnumber;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an `int`, the second one to a `char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (`*`) that we use when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator that we have seen a bit earlier, but which is also written with an asterisk (`*`). They are simply two different things represented with the same sign.

© cplusplus.com 2008. All rights reserved

Now have a look at this code:

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Notice that even though we have never directly set a value to either `firstvalue` or `secondvalue`, both end up with a value set indirectly through the use of `mypointer`. This is the procedure:

First, we have assigned as value of `mypointer` a reference to `firstvalue` using the reference operator (`&`). And then we have assigned the value 10 to the memory location pointed by `mypointer`, that because at this moment is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with `secondvalue` and that same pointer, `mypointer`.

Here is an example a little bit more elaborated:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by
p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

I have included as a comment on each line how the code can be read: ampersand (`&`) as "address of" and asterisk (`*`) as "value pointed by".

Notice that there are expressions with pointers `p1` and `p2`, both with and without dereference operator (`*`). The meaning of an expression using the dereference operator (`*`) is very different from one that does not: When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

© cplusplus.com 2008. All rights reserved

Another thing that may call your attention is the line:

```
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type `int*` (pointer to `int`).

Otherwise, the type for the second variable declared in that line would have been `int` (and not `int*`) because of precedence relationships. If we had written:

```
int * p1, p2;
```

`p1` would indeed have `int*` type, but `p2` would have type `int` (spaces do not matter at all for this purpose). This is due to operator precedence rules. But anyway, simply remembering that you have to put one asterisk per pointer is enough for most pointer users.

Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

```
int numbers [20];
int * p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

```
numbers = p;
```

Because `numbers` is an array, so it operates as a constant pointer, and we cannot assign values to constants.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

© cplusplus.com 2008. All rights reserved

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

In the chapter about arrays we used brackets (`[]`) several times in order to specify the index of an element of the array to which we wanted to refer. Well, these bracket sign operators `[]` are also a dereference operator known as *offset operator*. They dereference the variable they follow just as `*` does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

These two expressions are equivalent and valid both if `a` is a pointer or if `a` is an array.

Pointer initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:

```
int number;
int *tommy = &number;
```

The behavior of this code is equivalent to:

```
int number;
int *tommy;
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (`tommy`), never the value being pointed (`*tommy`). You must consider that at the moment of declaring a pointer, the asterisk (`*`) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

```
int number;
int *tommy;
*tommy = &number;
```

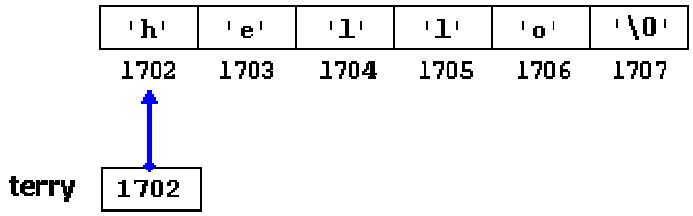
that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
char * terry = "hello";
```


© cplusplus.com 2008. All rights reserved

In this case, memory space is reserved to contain "hello" and then a pointer to the first character of this memory block is assigned to terry. If we imagine that "hello" is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:



It is important to indicate that terry contains the value 1702, and not 'h' nor "hello", although 1702 indeed is the address of both of these.

The pointer terry points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these two expression:

```
*(terry+4)
terry[4]
```

Both expressions have a value of 'o' (the fifth element of the array).

Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4.

Suppose that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

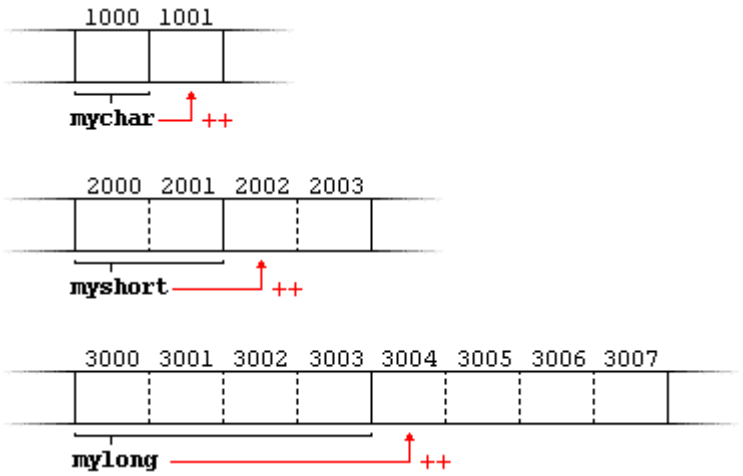
and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:

```
mychar++;
myshort++;
mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.

© cplusplus.com 2008. All rights reserved



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Both the increase (`++`) and decrease (`--`) operators have greater operator precedence than the dereference operator (`*`), but both have a special behavior when used as suffix (the expression is evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++
```

Because `++` has greater precedence than `*`, this expression is equivalent to `*(p++)`. Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

```
(*p)++
```

Here, the expression would have been evaluated as the value pointed by `p` increased by one. The value of `p` (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

```
*p++ = *q++;
```

Because `++` has a higher precedence than `*`, both `p` and `q` are increased, but because both increase operators (`++`) are used as postfix and not prefix, the value assigned to `*p` is `*q` before both `p` and `q` are increased. And then both are increased. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

Like always, I recommend you to use parentheses `()` in order to avoid unexpected results and to give more legibility to the code.

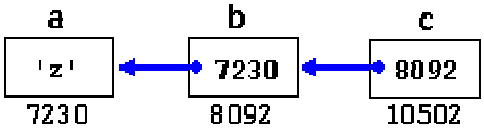
© cplusplus.com 2008. All rights reserved

Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (`*`) for each level of reference in their declarations:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of `7230`, `8092` and `10502`, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

The new thing in this example is variable `c`, which can be used in three different levels of indirection, each one of them would correspond to a different value:

- `c` has type `char**` and a value of `8092`
- `*c` has type `char*` and a value of `7230`
- `**c` has type `char` and a value of `'z'`

void pointers

The `void` type of pointer is a special type of pointer. In C++, `void` represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

One of its uses may be to pass generic parameters to a function:

© cplusplus.com 2008. All rights reserved

```
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

```
y, 1603
```

`sizeof` is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non-dynamic data types this value is a constant. Therefore, for example, `sizeof(char)` is `1`, because `char` type is one byte long.

Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;      // p has a null pointer value
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses `()` and an asterisk (`*`) is inserted before the name:

© cplusplus.com 2008. All rights reserved

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int
(*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

```
8
```

In the example, `minus` is a pointer to a function that has two parameters of type `int`. It is immediately assigned to point to the function `subtraction`, all in a single line:

```
int (* minus)(int,int) = subtraction;
```


© cplusplus.com 2008. All rights reserved

Dynamic Memory

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is *dynamic memory*, for which C++ integrates the operators `new` and `delete`.

Operators new and new[]

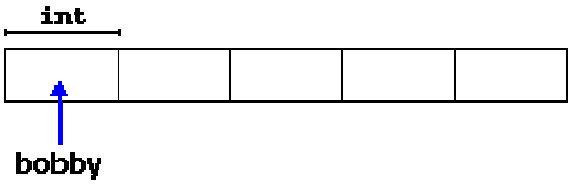
In order to request dynamic memory we use the operator `new`. `new` is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to assign a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:

```
int * bobby;
bobby = new int [5];
```

In this case, the system dynamically assigns space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `bobby`. Therefore, now, `bobby` points to a valid block of memory with space for five elements of type `int`.



The first element pointed by bobby can be accessed either with the expression `bobby[0]` or the expression `*bobby`. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with `bobby[1]` or `*(bobby+1)` and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type `bad_alloc` is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

© cplusplus.com 2008. All rights reserved

This exception method is the default method used by new, and is the one used in a declaration like:

```
bobby = new int [5];  // if it fails an exception is thrown
```

The other method is known as `nothrow`, and what happens when it is used is that when a memory allocation fails, instead of throwing a `bad_alloc` exception or terminating the program, the pointer returned by `new` is a null pointer, and the program continues its execution.

This method can be specified by using a special object called `nothrow`, declared in header `<new>`, as argument for `new`:

```
bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if `bobby` took a null pointer value:

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
  // error assigning memory. Take measures.
  };
```

This `nothrow` method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

Operators delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator `delete`, whose format is:

```
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with `new`, or a null pointer (in the case of a null pointer, `delete` produces no effect).

© cplusplus.com 2008. All rights reserved

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the `new` statement is a variable value entered by the user (`i`), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (`Error: memory could not be allocated`). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the `nothrow` method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

Dynamic memory in ANSI-C

Operators `new` and `delete` are exclusive of C++. They are not available in the C language. But using pure C language and its library, dynamic memory can also be used through the functions malloc, calloc, realloc and free, which are also available in C++ including the `<cstdlib>` header file (see cstdlib for more info).

The memory blocks allocated by these functions are not necessarily compatible with those returned by new, so each one should be manipulated with its own set of functions or operators.

© cplusplus.com 2008. All rights reserved

Data structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not mere sequences of elements all of the same data type, but sets of different elements with different data types.

Data structures

A data structure is a group of data elements grouped together under one name. These data elements, known as *members*, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```
struct structure_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

where `structure_name` is a name for the structure type, `object_name` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{ }` there is a list with the data members, each one is specified with a type and a valid identifier as its name.

The first thing we have to know is that a data structure creates a new type: Once a data structure is declared, a new type with the identifier specified as `structure_name` is created and can be used in the rest of the program as if it was any other type. For example:

```
struct product {
  int weight;
  float price;
} ;

product apple;
product banana, melon;
```

We have first declared a structure type called `product` with two members: `weight` and `price`, each of a different fundamental type. We have then used this name of the structure type (`product`) to declare three objects of that type: `apple`, `banana` and `melon` as we would have done with any fundamental data type.

Once declared, `product` has become a new valid type name like the fundamental ones `int`, `char` or `short` and from that point on we are able to declare objects (variables) of this compound new type, like we have done with `apple`, `banana` and `melon`.

Right at the end of the `struct` declaration, and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `banana` and `melon` at the moment we define the data structure type this way:

```
struct product {
  int weight;
  float price;
} apple, banana, melon;
```

It is important to clearly differentiate between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like `apple`, `banana` and `melon`) from a single structure type (`product`).

© cplusplus.com 2008. All rights reserved

Once we have declared our three objects of a determined structure type (`apple`, `banana` and `melon`) we can operate directly with their members. To do that we use a dot (`.`) inserted between the object name and the member name. For example, we could operate with any of these elements as if they were standard variables of their respective types:

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Each one of these has the data type corresponding to the member they refer to: `apple.weight`, `banana.weight` and `melon.weight` are of type `int`, while `apple.price`, `banana.price` and `melon.price` are of type `float`.

Let's see a real example where you can see how a structure type can be used in the same way as fundamental types:

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  string mystr;

  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;

  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;

  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

The example shows how we can use the members of an object as regular variables. For example, the member `yours.year` is a valid variable of type `int`, and `mine.title` is a valid variable of type `string`.

The objects `mine` and `yours` can also be treated as valid variables of type `movies_t`, for example we have passed them to the function `printmovie` as we would have done with regular variables. Therefore, one of the most important advantages of data structures is that we can either refer to their members individually or to the entire structure as a block with only one identifier.

© cplusplus.com 2008. All rights reserved

Data structures are a feature that can be used to represent databases, especially if we consider the possibility of building arrays of them:

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

#define N MOVIES 3

struct movies_t {
  string title;
  int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main ()
{
  string mystr;
  int n;

  for (n=0; n<N_MOVIES; n++)
  {
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
  }

  cout << "\nYou have entered these movies:\n";
  for (n=0; n<N_MOVIES; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

```
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
Matrix (1999)
Taxi Driver (1976)
```

Pointers to structures

Like any other type, structures can be pointed by its own type of pointers:

```
struct movies_t {
  string title;
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

Here `amovie` is an object of structure type `movies_t`, and `pmovie` is a pointer to point to objects of structure type `movies_t`. So, the following code would also be valid:

```
pmovie = &amovie;
```

The value of the pointer `pmovie` would be assigned to a reference to the object `amovie` (its memory address).

© cplusplus.com 2008. All rights reserved

We will now go with another example that includes pointers, which will serve to introduce a new operator: the arrow operator (->):

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;
};

int main ()
{
  string mystr;

  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;

  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;

  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";

  return 0;
}
```

```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

The previous code includes an important introduction: the arrow operator (->). This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used:

```
pmovie->title
```

Which is for all purposes equivalent to:

```
(*pmovie).title
```

Both expressions `pmovie->title` and `(*pmovie).title` are valid and both mean that we are evaluating the member `title` of the data structure pointed by a pointer called `pmovie`. It must be clearly differentiated from:

```
*pmovie.title
```

which is equivalent to:

```
*(pmovie.title)
```

And that would access the value pointed by a hypothetical pointer member called `title` of the structure object `pmovie` (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:

© cplusplus.com 2008. All rights reserved

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

Nesting structures

Structures can also be nested so that a valid element of a structure can also be in its turn another structure.

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
  } charlie, maria;

friends_t * pfriends = &charlie;
```

After the previous declaration we could use any of the following expressions:

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

(where, by the way, the last two expressions refer to the same member).

© cplusplus.com 2008. All rights reserved

Other Data Types

Defined data types (typedef)

C++ allows the definition of our own types based on other existing data types. We can do this using the keyword `typedef`, whose format is:

```
typedef existing_type new_type_name ;
```

where `existing_type` is a C++ fundamental or compound type and `new_type_name` is the name for the new type we are defining. For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

In this case we have defined four data types: `C`, `WORD`, `pChar` and `field` as `char`, `unsigned int`, `char*` and `char[50]` respectively, that we could perfectly use in declarations later as any other valid type:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

`typedef` does not create different types. It only creates synonyms of existing types. That means that the type of `myword` can be considered to be either `WORD` or `unsigned int`, since both are in fact the same type.

`typedef` can be useful to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

Unions

Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union union_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

All the elements of the `union` declaration occupy the same physical space in memory. Its size is the one of the greatest element of the declaration. For example:

© cplusplus.com 2008. All rights reserved

```
union mytypes_t {
  char c;
  int i;
  float f;
  } mytypes;
```

defines three elements:

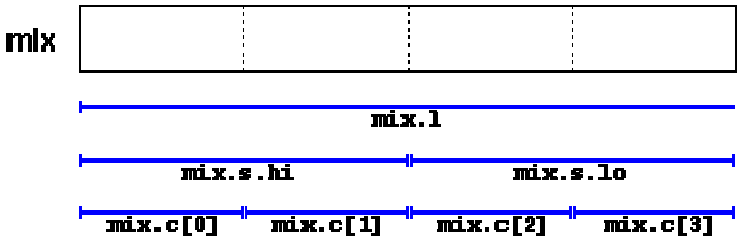
```
mytypes.c
mytypes.i
mytypes.f
```

each one with a different data type. Since all of them are referring to the same location in memory, the modification of one of the elements will affect the value of all of them. We cannot store different values in them independent of each other.

One of the uses a union may have is to unite an elementary type with an array or structures of smaller elements. For example:

```
union mix_t {
  long l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

defines three names that allow us to access the same group of 4 bytes: `mix.l`, `mix.s` and `mix.c` and which we can use according to how we want to access these bytes, as if they were a single `long`-type data, as if they were two `short` elements or as an array of `char` elements, respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data. For a *little-endian* system (most PC platforms), this union could be represented as:



The exact alignment and order of the members of a union in memory is platform dependant. Therefore be aware of possible portability issues with this type of use.

Anonymous unions

In C++ we have the option to declare anonymous unions. If we declare a union without any name, the union will be anonymous and we will be able to access its members directly by their member names. For example, look at the difference between these two structure declarations:

© cplusplus.com 2008. All rights reserved

structure with regular union	structure with anonymous union

struct {
  char title[50];
  char author[50];
  union {
  float dollars;
  int yens;
  } price;
} book;
	struct {
	  char title[50];
	  char author[50];
	  union {
	  float dollars;
	  int yens;
	  };
	} book;

The only difference between the two pieces of code is that in the first one we have given a name to the union (`price`) and in the second one we have not. The difference is seen when we access the members `dollars` and `yens` of an object of this type. For an object of the first type, it would be:

```
book.price.dollars
book.price.yens
```

whereas for an object of the second type, it would be:

```
book.dollars
book.yens
```

Once again I remind you that because it is a union and not a struct, the members `dollars` and `yens` occupy the same physical space in the memory so they cannot be used to store two different values simultaneously. You can set a value for `price` in dollars or in yens, but not in both.

Enumerations (enum)

Enumerations create new data types to contain something different that is not limited to the values fundamental data types may take. Its form is the following:

```
enum enumeration_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

For example, we could create a new type of variable called `color` to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

Notice that we do not include any fundamental data type in the declaration. To say it somehow, we have created a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type `color_t` may take are the new constant values included within braces. For example, once the `colors_t` enumeration is declared the following expressions will be valid:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

Enumerations are type compatible with numeric variables, so their constants are always assigned an integer numerical value internally. If it is not specified, the integer value equivalent to the first possible value is equivalent to `0` and the following ones follow a +1 progression. Thus, in our data type `colors_t` that we have defined above, `black` would be equivalent to `0`, `blue` would be equivalent to `1`, `green` to `2`, and so on.

© cplusplus.com 2008. All rights reserved

We can explicitly specify an integer value for any of the constant values that our enumerated type can take. If the constant value that follows it is not given an integer value, it is automatically assumed the same value as the previous one plus one. For example:

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

In this case, variable `y2k` of enumerated type `months_t` can contain any of the 12 possible values that go from `january` to `december` and that are equivalent to values between `1` and `12` (not between `0` and `11`, since we have made `january` equal to `1`).

© cplusplus.com 2008. All rights reserved

Object Oriented Programming

Classes (I)

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights that the members following them acquire:

- `private` members of a class are accessible only from within other members of the same class or from their *friends*.
- `protected` members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, `public` members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the `class` keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Declares a class (i.e., a type) called `CRectangle` and an object (i.e., a variable) of this class called `rect`. This class contains four members: two data members of type `int` (member `x` and member `y`) with private access (because private is the default access level) and two member functions with public access: `set_values()` and `area()`, of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, `CRectangle` was the class name (i.e., the type), whereas rect was an object of type `CRectangle`. It is the same relationship `int` and `a` have in the following declaration:

© cplusplus.com 2008. All rights reserved

```
int a;
```

where `int` is the type name (the class) and `a` is the variable name (the object).

After the previous declarations of `CRectangle` and `rect`, we can refer within the body of the program to any of the public members of the object `rect` as if they were normal functions or normal variables, just by putting the object's name followed by a dot (`.`) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are `x` and `y`, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

The most important new thing in this code is the operator of scope (`::`, two colons) included in the definition of `set_values()`. It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function `area()` has been included directly within the definition of the `CRectangle` class given its extreme simplicity, whereas `set_values()` has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (`::`) to specify that we are defining a function that is a member of the class `CRectangle` and not a regular global function.

The scope operator (`::`) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function `set_values()` of the previous code, we have been able to use the variables `x` and `y`, which are private members of class `CRectangle`, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members `x` and `y` have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the

© cplusplus.com 2008. All rights reserved

class. This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class `CRectangle`, we could have declared the object `rectb` in addition to the object `rect`:

```
// example: one class, two objects
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect, rectb;
  rect.set values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is `CRectangle`, of which there are two instances or objects: `rect` and `rectb`. Each one of them has its own member variables and member functions.

Notice that the call to `rect.area()` does not give the same result as the call to `rectb.area()`. This is because each object of class CRectangle has its own variables `x` and `y`, as they, in some way, have also their own function members `set_value()` and `area()` that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to `rect.area` or `rectb.area`. Those member functions directly used the data members of their respective objects `rect` and `rectb`.

Constructors and destructors

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function `area()` before having called function `set_values()`? Probably we would have gotten an undetermined result since the members `x` and `y` would have never been assigned a value.

In order to avoid that, a class can include a special function called `constructor`, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even `void`.

We are going to implement `CRectangle` including a constructor:

© cplusplus.com 2008. All rights reserved

```
// example: class constructor
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function `set_values()`, and have included instead a constructor that performs a similar action: it initializes the values of `x` and `y` with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even `void`.

The *destructor* fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (`~`) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

© cplusplus.com 2008. All rights reserved

```
// example on constructors and destructors
#include <iostream>
using namespace std;

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

© cplusplus.com 2008. All rights reserved

```
// overloading class constructors
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12
rectb area: 25
```

In this case, `rectb` was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both `width` and `height` with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses `()`:

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

The compiler assumes that `CExample` has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

© cplusplus.com 2008. All rights reserved

```
class CExample {
  public:
    int a,b,c;
    CExample (int n, int m) { a=n; b=m; };
    void multiply () { c=a*b; };
  };
```

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For `CExample`, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);
CExample ex2 (ex);   // copy constructor (data copied from ex)
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class `CRectangle`.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (`->`) of indirection. Here is an example with some possible combinations:

© cplusplus.com 2008. All rights reserved

```
// pointer to classes example
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set values (int, int);
    int area (void) {return (width * height);}
};

void CRectangle::set values (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() << endl;
  cout << "d[1] area: " << d[1].area() << endl;
  delete[] d;
  delete b;
  return 0;
}
```

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

Next you have a summary on how can you read some pointer and class operators (`*`, `&`, `.`, `->`, `[ ]`) that appear in the previous example:

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Be sure that you understand the logic under all of these expressions before proceeding with the next sections. If you have doubts, read again this section and/or consult the previous sections about pointers and data structures.

Classes defined with struct and union

Classes can be defined not only with keyword `class`, but also with keywords `struct` and `union`.

The concepts of class and data structure are so similar that both keywords (`struct` and `class`) can be used in C++ to declare classes (i.e. `struct`s can also have function members in C++, not only data members). The only difference between both is that members of classes declared with the keyword `struct` have public access by default, while members of classes declared with the keyword `class` have private access. For all other purposes both keywords are equivalent.

© cplusplus.com 2008. All rights reserved

The concept of unions is different from that of classes declared with `struct` and `class`, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold function members. The default access in union classes is public.

© cplusplus.com 2008. All rights reserved

Classes (II)

Overloading operators

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

```
int a, b, c;
a = b + c;
```

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

```
struct {
  string product;
  float price;
} a, b, c;
a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the behavior our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the `operator` keyword followed by the operator sign that we want to overload. The format is:

```
type operator sign (parameters) { /*...*/ }
```

Here you have an example that overloads the addition operator (`+`). We are going to create a class to store bidimensional vectors and then we are going to add two of them: `a(3,1)` and `b(1,2)`. The addition of two bidimensional vectors is an operation as simple as adding the two `x` coordinates to obtain the resulting `x` coordinate and adding the two `y` coordinates to obtain the resulting `y`. In this case the result will be `(3+1,1+2) = (4,3)`.

© cplusplus.com 2008. All rights reserved

```
// vectors: overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

```
4,3
```

It may be a little confusing to see so many times the `CVector` identifier. But, consider that some of them refer to the class name (type) `CVector` and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int);             // function name CVector (constructor)
CVector operator+ (CVector);    // function returns a CVector
```

The function `operator+` of class `CVector` is the one that is in charge of overloading the addition operator (`+`). This function can be called either implicitly using the operator, or explicitly using the function name:

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector () { };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

© cplusplus.com 2008. All rights reserved

```
CVector c;
```

included in `main()` would not have been valid.

Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables `x` and `y` undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code I have not included in the example.

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (`=`) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3);
CVector e;
e = d;           // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use `operator +` to subtract two classes or `operator==` to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function `operator+` can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

Expression	Operator	Member function	Global function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@ (B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@ (B)	-
a(b, c...)	()	A::operator() (B, C...)	-
a->x	->	A::operator->()	-

Where `a` is an object of class `A`, `b` is an object of class `B` and `c` is an object of class `C`.

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

The keyword this

The keyword `this` represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.

© cplusplus.com 2008. All rights reserved

One of its uses can be to check if a parameter passed to a member function is the object itself. For example,

```
// this
#include <iostream>
using namespace std;

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

```
yes, &a is b
```

It is also frequently used in `operator=` member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an `operator=` function similar to this one:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

In fact this function is very similar to the code that the compiler generates implicitly for this class if we do not include an `operator=` member function to copy objects of this class.

Static members

A class can contain *static* members, either data or functions.

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

© cplusplus.com 2008. All rights reserved

```
// static members in classes
#include <iostream>
using namespace std;

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

```
7
6
```

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

```
int CDummy::n=0;
```

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```
cout << a.n;
cout << CDummy::n;
```

These two calls included in the previous example are referring to the same variable: the static variable `n` within class `CDummy` shared by all objects of this class.

Once again, I remind you that in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.

Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword `this`, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.

© cplusplus.com 2008. All rights reserved

Friendship and inheritance

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.

Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword `friend`:

```
// friend functions
#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area () {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
  return 0;
}
```

```
24
```

The `duplicate` function is a friend of `CRectangle`. From within that function we have been able to access the members `width` and `height` of different objects of type `CRectangle`, which are private members. Notice that neither in the declaration of `duplicate()` nor in its later use in `main()` have we considered `duplicate` a member of class `CRectangle`. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate `duplicate()` within the class `CRectangle`.

© cplusplus.com 2008. All rights reserved

Friend classes

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
// friend class
#include <iostream>
using namespace std;

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

```
16
```

In this example, we have declared `CRectangle` as a friend of `CSquare` so that `CRectangle` member functions could have access to the protected and private members of `CSquare`, more concretely to `CSquare::side`, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class `CSquare`. This is necessary because within the declaration of `CRectangle` we refer to CSquare (as a parameter in `convert()`). The definition of `CSquare` is included later, so if we did not include a previous empty declaration for `CSquare` this class would not be visible from within the definition of `CRectangle`.

Consider that friendships are not corresponded if we do not explicitly specify so. In our example, `CRectangle` is considered as a friend class by `CSquare`, but `CRectangle` does not consider `CSquare` to be a friend, so `CRectangle` can access the protected and private members of `CSquare` but not the reverse way. Of course, we could have declared also `CSquare` as friend of `CRectangle` if we wanted to.

Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

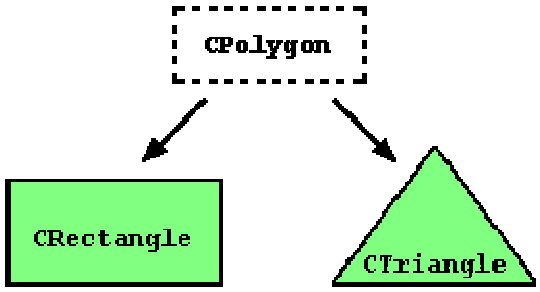
Inheritance between classes

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going

© cplusplus.com 2008. All rights reserved

to suppose that we want to declare a series of classes that describe polygons like our `CRectangle`, or like `CTriangle`. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class `CPolygon` from which we would derive the two other ones: `CRectangle` and `CTriangle`.



The class `CPolygon` would contain members that are common for both types of polygon. In our case: `width` and `height`. And `CRectangle` and `CTriangle` would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member `A` and we derive it to another class with another member called `B`, the derived class will contain both members `A` and `B`.

In order to derive a class from another, we use a colon (`:`) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where `derived_class_name` is the name of the derived class and `base_class_name` is the name of the class on which it is based. The `public` access specifier may be replaced by any one of the other access specifiers `protected` and `private`. This access specifier describes the minimum access level for the members that are inherited from the base class.

© cplusplus.com 2008. All rights reserved

```
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

```
20
10
```

The objects of the classes `CRectangle` and `CTriangle` each contain members inherited from `CPolygon`. These are: `width`, `height` and `set_values()`.

The `protected` access specifier is similar to `private`. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted `width` and `height` to be accessible from members of the derived classes `CRectangle` and `CTriangle` and not only by members of `CPolygon`, we have used `protected` access instead of `private`.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from `main()`, from another class or from a function.

In our example, the members inherited by `CRectangle` and `CTriangle` have the same access permissions as they had in their base class `CPolygon`:

© cplusplus.com 2008. All rights reserved

```
CPolygon::width           // protected access
CRectangle::width         // protected access

CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

This is because we have used the `public` keyword to define the inheritance relationship on each of the derived classes:

```
class CRectangle: public CPolygon { ... }
```

This `public` keyword after the colon (`:`) denotes the maximum access level for all the members inherited from the class that follows it (in this case `CPolygon`). Since `public` is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like `protected`, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: `private`, all the base class members are inherited as private.

For example, if `daughter` was a class derived from `mother` that we defined as:

```
class daughter: protected mother;
```

This would set `protected` as the maximum access level for the members of `daughter` that it inherited from `mother`. That is, all members that were public in `mother` would become protected in `daughter`. Of course, this would not restrict `daughter` to declare its own public members. That maximum access level is only set for the members inherited from `mother`.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with `class` keyword and public for those declared with `struct`.

What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
```

For example:

© cplusplus.com 2008. All rights reserved

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (0);
  son daniel(0);

  return 0;
}
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Notice the difference between which `mother`'s constructor is called when a new `daughter` object is created and which when it is a `son` object. The difference is because the constructor declaration of `daughter` and `son`:

```
daughter (int a)          // nothing specified: call default
son (int a) : mother (a)  // constructor specified: call this
```

Multiple inheritance

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (`COutput`) and we wanted our classes `CRectangle` and `CTriangle` to also inherit its members in addition to those of `CPolygon` we could write:

```
class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

here is the complete example:

© cplusplus.com 2008. All rights reserved

```
// multiple inheritance
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);
  };

void COutput::output (int i) {
  cout << i << endl;
  }

class CRectangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

```
20
10
```


© cplusplus.com 2008. All rights reserved

Polymorphism

Before getting into this section, it is recommended that you have a proper understanding of pointers and class inheritance. If any of the following statements seem strange to you, you should review the indicated sections:

Statement:	Explained in:
int a::b(c) {};	Classes
a->b	Data Structures
class a: public b;	Friendship and inheritance

Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

We are going to start by rewriting our program about the rectangle and the triangle of the previous section taking into consideration this pointer compatibility property:

```
// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

```
20
10
```

In function `main`, we create two pointers that point to objects of class `CPolygon` (`ppoly1` and `ppoly2`). Then we assign references to `rect` and `trgl` to these pointers, and because both are objects of classes derived from `CPolygon`, both are valid assignment operations.

The only limitation in using `*ppoly1` and `*ppoly2` instead of `rect` and `trgl` is that both `*ppoly1` and `*ppoly2` are of type `CPolygon*` and therefore we can only use these pointers to refer to the members that `CRectangle` and

© cplusplus.com 2008. All rights reserved

CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

Virtual members

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return (0); }
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

```
20
10
0
```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

© cplusplus.com 2008. All rights reserved

Therefore, what the `virtual` keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of its virtuality, we have also been able to declare an object of type `CPolygon` and to call its own `area()` function, which always returns 0.

Abstract base classes

Abstract base classes are something very similar to our `CPolygon` class of our previous example. The only difference is that in our previous example we have defined a valid `area()` function with a minimal functionality for objects that were of class `CPolygon` (like the object `poly`), whereas in an abstract base classes we could leave that `area()` member function without implementation at all. This is done by appending `=0` (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```
// abstract class CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Notice how we appended `=0` to `virtual int area ()` instead of specifying an implementation for the function. This type of function is called a *pure virtual function*, and all classes that contain at least one pure virtual function are *abstract base classes*.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

```
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

```
CPolygon * ppoly1;
CPolygon * ppoly2;
```

would be perfectly valid.

This is so for as long as `CPolygon` includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes.

Here you have the complete example:

© cplusplus.com 2008. All rights reserved

```
// abstract base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

```
20
10
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (`CPolygon*`). This can be tremendously useful. For example, now we can create a function member of the abstract base class `CPolygon` that is able to print on screen the result of the `area()` function even though `CPolygon` itself has no implementation for this function:

© cplusplus.com 2008. All rights reserved

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

```
20
10
```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:

© cplusplus.com 2008. All rights reserved

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void)
      { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

```
20
10
```

Notice that the `ppoly` pointers:

```
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to `CPolygon` but the objects dynamically allocated have been declared having the derived class type directly.

© cplusplus.com 2008. All rights reserved

Advanced concepts

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword `class` or the keyword `typename`. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b);
}
```

Here we have created a template function with `myType` as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template `GetMax` returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call `GetMax` to compare two integer values of type `int` we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of `myType` by the type passed as the actual template parameter (`int` in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

© cplusplus.com 2008. All rights reserved

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

In this case, we have used `T` as the template parameter name instead of `myType` because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template `GetMax()` twice. The first time with arguments of type `int` and the second one with arguments of type `long`. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type `T` is used within the `GetMax()` template function even to declare new objects of that type:

```
T result;
```

Therefore, `result` will be an object of the same type as the parameters `a` and `b` when the function template is instantiated with a specific type.

In this specific case where the generic type `T` is used as a parameter for `GetMax` the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying `<int>` and `<long>`). So we could have written instead:

```
int i,j;
GetMax (i,j);
```

Since both `i` and `j` are of type `int`, and the compiler can automatically find out that the template parameter can only be `int`. This implicit method produces exactly the same result:

© cplusplus.com 2008. All rights reserved

```
// function template II
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

Notice how in this case, we called our function template `GetMax()` without explicitly specifying the type between angle-brackets `<>`. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (`class T`) and the function template itself accepts two parameters, both of this `T` type, we cannot call our function template with two objects of different types as arguments:

```
int i;
long l;
k = GetMax (i,l);
```

This would not be correct, since our `GetMax` function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template `GetMin()` accepts two parameters of different types and returns an object of the same type as the first parameter (`T`) that is passed. For example, after that declaration we could call `GetMin()` with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though `j` and `l` have different types, since the compiler can determine the appropriate instantiation anyway.

© cplusplus.com 2008. All rights reserved

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the `template <...>` prefix:

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

```
100
```

Notice the syntax of the definition of member function getmax:

© cplusplus.com 2008. All rights reserved

```
template <class T>
T mypair<T>::getmax ()
```

Confused by so many `T`'s? There are three `T`'s in this declaration: The first one is the template parameter. The second `T` refers to the type returned by the function. And the third `T` (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called `mycontainer` that can store one element of any type and that it has just one member function called `increase`, which increases its value. But we find that when it stores an element of type `char` it would be more convenient to have a completely different implementation with a function member `uppercase`, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

```
8
J
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty`template<>` parameter list. This is to explicitly declare it as a template specialization.

© cplusplus.com 2008. All rights reserved

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
using namespace std;

template <class T, int N>
class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};

template <class T, int N>
void mysequence<T,N>::setmember (int x, T value)
{
  memblock[x]=value;
}

template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}

int main () {
  mysequence <int,5> myints;
  mysequence <double,5> myfloats;
  myints.setmember (0,100);
  myfloats.setmember (3,3.1416);
  cout << myints.getmember(0) << '\n';
  cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

```
100
3.1416
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:

© cplusplus.com 2008. All rights reserved

```
mysequence<char,10> myseq;
```

Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.

When projects grow it is usual to split the code of a program in different source code files. In these cases, the interface and implementation are generally separated. Taking a library of functions as example, the interface generally consists of declarations of the prototypes of all the functions that can be called. These are generally declared in a "header file" with a .h extension, and the implementation (the definition of these functions) is in an independent file with c++ code.

Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.

© cplusplus.com 2008. All rights reserved

Namespaces

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

```
namespace identifier
{
entities
}
```

Where `identifier` is any valid identifier and `entities` is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`. In order to access these variables from outside the `myNamespace` namespace we have to use the scope operator `::`. For example, to access the previous variables from outside `myNamespace` we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

```
// namespaces
#include <iostream>
using namespace std;

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

```
5
3.1416
```

In this case, there are two global variables with the same name: `var`. One is defined within the namespace `first` and the other one in `second`. No redefinition errors happen thanks to namespaces.

© cplusplus.com 2008. All rights reserved

using

The keyword `using` is used to introduce a name from a namespace into the current declarative region. For example:

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;
  return 0;
}
```

```
5
2.7183
10
3.1416
```

Notice how in this code, `x` (without any name qualifier) refers to `first::x` whereas `y` refers to `second::y`, exactly as our `using` declarations have specified. We still have access to `first::y` and `second::x` using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << endl;
  cout << y << endl;
  cout << second::x << endl;
  cout << second::y << endl;
  return 0;
}
```

```
5
10
3.1416
2.7183
```

In this case, since we have declared that we were `using namespace first`, all direct uses of `x` and `y` without name qualifiers were referring to their declarations in `namespace first`.

© cplusplus.com 2008. All rights reserved

`using` and `using namespace` have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

```
// using namespace example
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << endl;
  }
  {
    using namespace second;
    cout << x << endl;
  }
  return 0;
}
```

```
5
3.1416
```

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

```
namespace new_name = current_name;
```

Namespace std

All the files in the C++ standard library declare all of its entities within the `std` namespace. That is why we have generally included the `using namespace std;` statement in all programs that used any entity defined in `iostream`.

© cplusplus.com 2008. All rights reserved

Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the try block:

```
// exceptions
#include <iostream>
using namespace std;

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. "
    cout << "Exception Nr. " << e << endl;
  }
  return 0;
}
```

```
An exception occurred. Exception Nr. 20
```

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value `20`), which is passed as an argument to the exception handler.

The exception handler is declared with the `catch` keyword. As you can see, it follows immediately the closing brace of the `try` block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (`...`) as the parameter of `catch`, that handler will catch any exception no matter what the type of the `throw` exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

© cplusplus.com 2008. All rights reserved

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

In this case the last handler would catch any exception thrown with any parameter that is neither an `int` nor a `char`.

After an exception has been handled the program execution resumes after the `try-catch` block, not after the `throw` statement!.

It is also possible to nest `try-catch` blocks within more external `try` blocks. In these cases, we have the possibility that an internal `catch` block forwards the exception to its external level. This is done with the expression `throw;` with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Exception specifications

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a `throw` suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called `myfunction` which takes one agument of type `char` and returns an element of type `float`. The only exception that this function might throw is an exception of type `int`. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular `int`-type handler.

If this `throw` specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no `throw` specifier (regular functions) are allowed to throw exceptions with any type:

```
int myfunction (int param) throw(); // no exceptions allowed
int myfunction (int param);         // all exceptions allowed
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called `exception` and is defined in the `<exception>` header file under the `namespace std`. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called `what` that returns a null-terminated character sequence (`char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.

© cplusplus.com 2008. All rights reserved

```
// standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << endl;
  }
  return 0;
}
```

```
My exception happened.
```

We have placed a handler that catches exception objects by reference (notice the ampersand `&` after the type), therefore this catches also classes derived from `exception`, like our `myex` object of class `myexception`.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this `std::exception` class. These are:

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator `new` and the memory cannot be allocated, an exception of type `bad_alloc` is thrown:

```
try
{
  int * myarray= new int[1000];
}
catch (bad alloc&)
{
  cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a `bad_alloc` exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion `int`s threw a `bad_alloc` exception.

Because `bad_alloc` is derived from the standard base class `exception`, we can handle that same exception by catching references to the `exception` class:

© cplusplus.com 2008. All rights reserved

```
// bad alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what()
<< endl;
  }
  return 0;
}
```


© cplusplus.com 2008. All rights reserved

Type Casting

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of `a` has been promoted from `short` to `int` and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (`short` to `int`, `int` to `float`, `double` to `int`...), to or from `bool`, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Here, a implicit conversion happened between objects of `class A` and `class B`, because `B` has a constructor that takes an object of class `A` as parameter. Therefore implicit conversions from `A` to `B` are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

© cplusplus.com 2008. All rights reserved

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
    float i,j;
};

class CAddition {
          int x,y;
  public:
          CAddition (int a, int b) { x=a; y=b; }
          int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to `CAddition`, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member `result` will produce either a run-time error or a unexpected result.

In order to control these types of conversions between classes, we have four specific casting operators: `dynamic_cast`, `reinterpret_cast`, `static_cast` and `const_cast`. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

The traditional type-casting equivalents to these expressions would be:

```
(new_type) expression
new_type (expression)
```

but each one with its own special characteristics:

dynamic_cast

`dynamic_cast` can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, `dynamic_cast` is always successful when we cast a class to one of its base classes:

© cplusplus.com 2008. All rights reserved

```
class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with `dynamic_cast` unless the base class is polymorphic.

When a class is polymorphic, `dynamic_cast` performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };

int main () {
  try {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;

    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;

    pd = dynamic cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}
```

```
Null pointer on second type-
cast
```

Compatibility note: `dynamic_cast` requires the Run-Time Type Information (RTTI) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using `dynamic_cast` to work properly.

The code tries to perform two dynamic casts from pointer objects of type `CBase*` (`pba` and `pbb`) to a pointer object of type `CDerived*`, but only the first one is successful. Notice their respective initializations:

```
CBase * pba = new CDerived;
CBase * pbb = new CBase;
```

Even though both are pointers of type `CBase*`, `pba` points to an object of type `CDerived`, while `pbb` points to an object of type `CBase`. Thus, when their respective type-castings are performed using `dynamic_cast`, `pba` is pointing to a full object of class `CDerived`, whereas `pbb` is pointing to an object of class `CBase`, which is an incomplete object of class `CDerived`.

When `dynamic_cast` cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead.

`dynamic_cast` can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (`void*`).

© cplusplus.com 2008. All rights reserved

static_cast

`static_cast` can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

`static_cast` can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

`reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` have no specific uses in C++ are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

© cplusplus.com 2008. All rights reserved

```
// const cast
#include <iostream>
using namespace std;

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "sample text";
  print ( const cast<char *> (c) );
  return 0;
}
```

```
sample text
```

typeid

`typeid` allows to check the type of an expression:

```
typeid (expression)
```

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header file `<typeinfo>`. This returned value can be compared with another one using operators `==` and `!=` or can serve to obtain a null-terminated character sequence representing the data type or class name by using its `name()` member.

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b are of different types:
a is: int *
b is: int
```

When `typeid` is applied to classes `typeid` uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

© cplusplus.com 2008. All rights reserved

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class CBase { virtual void f(){} };
class CDerived : public CBase {};

int main () {
  try {
    CBase* a = new CBase;
    CBase* b = new CDerived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << endl;
}
  return 0;
}
```

```
a is: class CBase *
b is: class CBase *
*a is: class CBase
*b is: class CDerived
```

Notice how the type that `typeid` considers for pointers is the pointer type itself (both `a` and `b` are of type `class CBase *`). However, when `typeid` is applied to objects (like `*a` and `*b`) `typeid` yields their dynamic type (i.e. the type of their most derived complete object).

If the type `typeid` evaluates is a pointer preceded by the dereference operator (`*`), and this pointer has a null value, `typeid` throws a `bad_typeid` exception.

© cplusplus.com 2008. All rights reserved

Preprocessor directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. These lines are always preceded by a hash sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

macro definitions (#define, #undef)

To define preprocessor macros we can use `#define`. Its format is:

```
#define identifier replacement
```

When the preprocessor encounters this directive, it replaces any occurrence of `identifier` in the rest of the code by `replacement`. This `replacement` can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++, it simply replaces any occurrence of `identifier` by `replacement`.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

After the preprocessor has replaced `TABLE_SIZE`, the code becomes equivalent to:

```
int table1[100];
int table2[100];
```

This use of #define as constant definer is already known by us from previous tutorials, but `#define` can work also with parameters to define function macros:

```
#define getmax(a,b) a>b?a:b
```

This would replace any occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier, exactly as you would expect if it was a function:

```
// function macro
#include <iostream>
using namespace std;

#define getmax(a,b) ((a)>(b)?(a):(b))

int main()
{
  int x=5, y;
  y= getmax(x,2);
  cout << y << endl;
  cout << getmax(7,x) << endl;
  return 0;
}
```

```
5
7
```

Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

© cplusplus.com 2008. All rights reserved

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

This would generate the same code as:

```
int table1[100];
int table2[200];
```

Function macro definitions accept two special operators (`#` and `##`) in the replacement sequence:
If the operator `#` is used before a parameter is used in the replacement sequence, that parameter is replaced by a string literal (as if it were enclosed between double quotes)

```
#define str(x) #x
cout << str(test);
```

This would be translated into:

```
cout << "test";
```

The operator `##` concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a ## b
glue(c,out) << "test";
```

This would also be translated into:

```
cout << "test";
```

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature, but be careful: code that relies heavily on complicated macros may result obscure to other programmers, since the syntax they expect is on many occasions different from the regular expressions programmers expect in C++.

Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met.

`#ifdef` allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

In this case, the line of code `int table[TABLE_SIZE];` is only compiled if `TABLE_SIZE` was previously defined with `#define`, independently of its value. If it was not defined, that line will not be included in the program compilation.

`#ifndef` serves for the exact opposite: the code between `#ifndef` and `#endif` directives is only compiled if the specified identifier has not been previously defined. For example:

© cplusplus.com 2008. All rights reserved

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the `TABLE_SIZE` macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the `#define` directive would not be executed.

The `#if`, `#else` and `#elif` (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` or `#elif` can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

Notice how the whole structure of `#if`, `#elif` and `#else` chained directives ends with `#endif`.

The behavior of `#ifdef` and `#ifndef` can also be achieved by using the special operators `defined` and `!defined` respectively in any `#if` or `#elif` directive:

```
#if !defined TABLE_SIZE
#define TABLE_SIZE 100
#elif defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
int table[TABLE_SIZE];
```

Line control (#line)

When we compile a program and some error happen during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

```
#line number "filename"
```

Where `number` is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

`"filename"` is an optional parameter that allows to redefine the file name that will be shown. For example:

© cplusplus.com 2008. All rights reserved

```
#line 20 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 20.

Error directive (#error)

This directive aborts the compilation process when it is found, generating a compilation the error that can be specified as its parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name __cplusplus is not defined (this macro name is defined by default in all C++ compilers).

Source file inclusion (#include)

This directive has also been used assiduously in other sections of this tutorial. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

```
#include "file"
#include <file>
```

The only difference between both expressions is the places (directories) where the compiler is going to look for the file. In the first case where the file name is specified between double-quotes, the file is searched first in the same directory that includes the file containing the directive. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.
If the file name is enclosed between angle-brackets <> the file is searched directly where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other specific header files are included using quotes.

Pragma directive (#pragma)

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.

If the compiler does not support a specific argument for #pragma, it is ignored - no error is generated.

Predefined macro names

The following macro names are defined at any time:

macro	value
__LINE__	Integer value representing the current line in the source code file being compiled.
__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__cplusplus	An integer value. All C++ compilers have this constant defined to some value. If the compiler is fully compliant with the C++ standard its value is equal or greater than 199711L depending on the version of the standard they comply.

For example:

© cplusplus.com 2008. All rights reserved

```
// standard macro names
#include <iostream>
using namespace std;

int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " <<   FILE   << ".\n";
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a __cplusplus value of ";
  cout <<   cplusplus;
  return 0;
}
```

```
This is the line number 7 of file
/home/jay/stdmacronames.cpp.
Its compilation began Nov  1 2005 at
10:12:29.
The compiler gives a __cplusplus value
of 1
```


© cplusplus.com 2008. All rights reserved

C++ Standard Library

Input/Output with files

C++ provides the following classes to perform output and input of characters to/from files:

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therfore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file
```

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

But let's go step by step:

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

© cplusplus.com 2008. All rights reserved

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```


© cplusplus.com 2008. All rights reserved

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Data input from a file can also be performed in the same way that we did with `cin`:

© cplusplus.com 2008. All rights reserved

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called `eof()` that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed `myfile.eof()` becomes true (i.e., the end of the file has been reached).

Checking state flags

In addition to `eof()`, which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

bad()

Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()

Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()

Returns true if a file open for reading has reached the end.

good()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function `clear()`, which takes no parameters.

get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer:

`ifstream`, like `istream`, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.

© cplusplus.com 2008. All rights reserved

ofstream, like ostream, has a pointer known as the *put pointer* that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

The other prototype for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off_type, which is also an integer type. And direction is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

© cplusplus.com 2008. All rights reserved

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Binary files

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like `getline` is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: `write` and `read`. The first one (`write`) is a member function of `ostream` inherited by `ofstream`. And `read` is a member function of `istream` that is inherited by `ifstream`. Objects of class `fstream` have both members. Their prototypes are:

```
write ( memory_block, size );
read ( memory_block, size );
```

Where `memory_block` is of type "pointer to char" (`char*`), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The `size` parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

```
// reading a complete binary file
#include <iostream>
#include <fstream>
using namespace std;

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.bin",
ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the complete file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
the complete file content is in memory
```

In this example the entire file is read and stored in a memory block. Let's examine how this is done:

© cplusplus.com 2008. All rights reserved

First, the file is open with the `ios::ate` flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member `tellg()`, we will directly obtain the size of the file. Notice the type we have used to declare variable `size`:

```
ifstream::pos type size;
```

`ifstream::pos_type` is a specific type used for buffer and file positioning and is the type returned by `file.tellg()`. This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use `int`:

```
int size;
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer of type `streambuf`. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an `ofstream`, each time the member function `put` (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called *synchronization* and takes place under any of the following circumstances:

- **When the file is closed:** before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: `flush` and `endl`.
- **Explicitly, with member function sync():** Calling stream's member function `sync()`, which takes no parameters, causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns `0`.

© cplusplus.com 2008. All rights reserved